Resolving Semantic Conflicts in Word Based Software Transactional Memory

Craig Sharp, William Blewitt, and Graham Morgan

Newcastle University, NE1 7RU, UK
{craig.sharp,william.blewitt,graham.morgan}@ncl.ac.uk

Abstract. In this paper we describe a technique for addressing semantic conflicts within word based Software Transactional Memory. A semantic conflict is considered to be some application condition which causes transactions to explicitly abort. Session locking and a companion Contention Management Policy are described which support the parallel exploration of multiple transaction schedules at run time, to resolve semantic conflicts. Performance figures are provided to demonstrate the effectiveness of our technique when semantic conflicts are introduced into established benchmarks.

Keywords: Transactional Memory, Contention Management, Shared Memory, Concurrency Control, STM.

1 Introduction

Software Transactional Memory (STM) has become a popular research area for concurrent programmers given that the STM abstraction offers ease of use in comparison to lock based approaches. More powerfully, composing sections of concurrent code can be achieved with ease using STM unlike a lock-based implementation [1]. At the time of writing, however, there exist a variety of STM implementations with two approaches gaining prominence: object based and word based. Object based STMs [2,3] are generally particular to object orientated languages and represent shared data in the form of *atomic objects*. Conversely, shared data in word based STMs [4,5] is represented at the level of *memory words*.

Felber et al observed in [4] that word based STMs allow transactional accesses to be mapped directly to the underlying memory system. As a result, word based STMs offer: (i) easier integration into existing programming languages and (ii) greater efficiency in the context of compiler support. *TinySTM* [4,6] has been provided as a lightweight and efficient word based STM. The (relatively) small code base makes *TinySTM* particularly attractive for STM development, allowing easy integration with the *STAMP* [7] benchmark suite. For these reasons, the developments in this paper have been integrated into *TinySTM*.

A significant feature of any STM system concerns the handling of aborted transactions under high contention for shared resources due to *concurrent* conflicts on shared data; a task typically delegated to the Contention Management

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 463–474, 2014.

Policy (CMP). Various CMPs exist to determine which transaction must abort upon a conflict (time-stamp CMP, for instance, gives priority to the transaction that began first). From the perspective of the application, however, there also exist *semantic* conflicts which can be conceived as conditions within the application which prevent a transaction from committing. Figure 1(A) provides an example of a semantic conflict with two threads executing a *withdrawal* and *deposit* transaction concurrently. Let us suppose that there is a concurrent conflict between the *withdrawer* and *depositor* transactions and that the CMP decides to abort the *depositor*. If the *withdrawer* requires that a deposit be made before it can perform the withdrawal, however, then it cannot continue and must (explicitly) abort. Both transactions re-execute until the *depositor* precedes the *withdrawer* (or the CMP aborts the *withdrawer*). If a CMP is employed which resolves conflicts based on transaction starting time or the amount of work completed, it is possible that the *withdrawer* may always succeed in aborting the *depositor* (if the *withdrawer* began before the *depositor* or has carried out more work, for instance).

Primitives exist to provide transaction coordination, which may in turn reduce the occurrence of semantic conflicts (e.g. Harris et al [1] provided primitives such as *retry* and *orElse*). Alternatively, a 'semantic transaction' can be avoided if simply allowed to commit rather than aborting explicitly (assuming no concurrent conflicts have occurred). The programmer may then specify that the transaction execute at some future time. Neither approach, however, alleviates the programmer from the burden of resolving the conflict. Conversely, [8] introduced a new CMP (*Hugh*) which resolves semantic conflicts without placing a burden on the programmer. *Hugh* was integrated with an object based STM and micro-benchmarks demonstrated some encouraging initial results. *Hugh2* has since been implemented with *TinySTM* (a word based STM). [9] describes the process of enabling transaction replication within *TinySTM* and severe implications for memory management are demonstrated (caused by the introduction of semantic conflicts). In this paper the following contributions are provided:

- The implementation of a novel *session* locking technique to resolve semantic conflicts in a manner both decoupled from the programmer and compatible with existing CMPs;
- Performance results showing the impact and resolution of semantic conflicts with several CMPs in large-scale benchmarks (e.g. *STAMP* benchmark suite [7]).

In Section 2 we describe the Implementation of our CMP and Section 3 summarises Related Work. Section 4 provides an Evaluation and, finally, Section 5 concludes the paper and summarises possible avenues for future work.

2 Implementation

2.1 Overview

Hugh2 CMP is activated once some $thread_x$ encounters a semantic conflict (causing $thread_x$ to explicitly abort its transaction). Before the aborted transaction

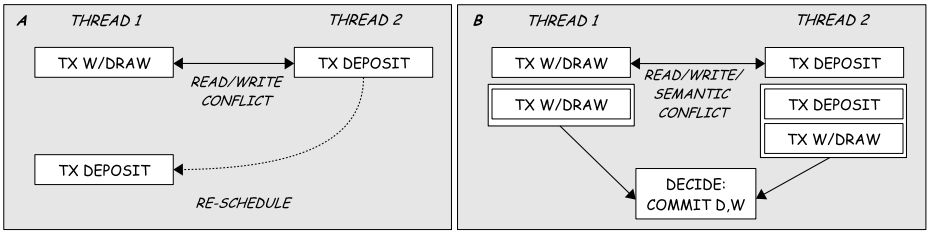

Fig. 1. Scenarios A and B contrast the approaches of a conventional CMP with the *Hugh2* CMP when a semantic conflict occurs

is restarted, $thread_x$ enters a new *session* mode. During *session* mode, $thread_x$ re-executes its own transaction in addition to the transactions of any other *session* mode threads. Each *session* mode thread executes a single permutation of transactions, to discover a schedule of transaction execution which resolves the semantic conflict(s). Figure 1(B) shows the re-execution of two transactions accessing a single account (specifically, a *depositor* and *withdrawer* transaction). Thread 2 executes a permutation which succeeds in resolving the semantic conflict (the deposit ensures that the withdrawal can occur).

When there are no further transactions to execute, each thread performs consensus to determine the permutation to be committed. Consensus is managed using a Universal Construction (hereafter UC). The UC is essentially a linked-list, which may be concurrently appended to by threads engaged in *session* mode. Each new entry of the UC identifies the transactions that have been committed during a particular *session*. Once a *session* has terminated each participating thread can determine whether its own transaction was committed or aborted by reading the log of the UC. Those threads whose transactions remain uncommitted perform a new *session*, while the threads of the committed transactions return to non-*session* mode.

2.2 Sessions

Hugh2 attempts to resolve semantic conflicts within the context of a *session*. A non-*session* mode thread will enter *session* mode if: (i) it encounters a semantic conflict while executing a transaction and (ii) it encounters data that is *session* locked while executing a transaction. In addition to the normal structures required by *TinySTM*, the following data structures are required to support *session* execution:

- A global *Transaction Table* is provided where the n-th entry into the table belongs to the n-th thread in the application. Threads in *session* mode retrieve and execute transactions stored in the table;
- A global UC is provided (a linked list) with a *session* counter (an integer). Each list entry corresponds to a *session* and the *session* counter identifies the

Algorithm 1. *TinySTM* Handlers

```
    function onStart(tx, ftn, args)
1       if tx.state ≠ started then return nocalltx;
2       if tx.nbAborts = 0 then setTableEntry(tx.id, ftn, args);
3       if tx.sessionMode then
4           setTableSession(tx.id, sessionCounter );
        else
5           return calltx;
6       while true do
7           if (txcall ← getNextTx(tx)) = noMoreTx then
8               if onTimeout(tx) = 0 then break else continue;
9           if consensusReached(sessionNo(tx.id)) then break;
10          Invoke(txcall.ftn, txcall.arg);
11          if onTxSuccess(tx, txcall) = 0 then break;
12      return nocalltx;
    function onPreCommit(tx)
13      if tx.state ≠ started then return;
14      logEntry ← UCLogEntry(sessionNo(tx.id));
15      if cas(&logEntry, logEntry, tx.txMask) = fail then
16          tx.state ← lostConsensus;
17          rollback();
        else
18          tx.state ← wonConsensus;
    function onCommit(tx)
19      logEntry ← UCLogEntry(sessionNo(tx.id));
20      if tx.state = wonConsensus then
21          atomicIncrement(sessionCounter);
22      tx.state ← started;
23      if bitIsSet(logEntry, tx.id) then
24          tx.sessionMode ← false;
        else
25          rollback();
```

current *session*. Every entry contains a bit mask denoting which transactions were committed for that particular *session*;

– Each thread also uses several variables to manage *session* execution including: a flag indicating whether it is in *session* mode, a state variable to hold its progress (which may hold the value: *started*, *lostConsensus* or *wonConsensus*), and a bit mask to record the transactions executed during a *session* (The i-th bit of the mask corresponds to the i-th entry in the *Transaction Table*).

TinySTM allows custom handlers to be called upon the occurrence of several important events during the per-thread execution of a transaction. *Hugh2* is mostly implemented within these handlers, specifically: *onStart*, *onPreCommit*, *onCommit* and *onAbort*. Algorithms 1 and 2 provide the pseudo code:

OnStart performs the iterative execution of transactions when a thread enters *session* mode. When a thread first executes a transaction it inserts the transaction function and argument to the transaction table (line 2). Non *session* mode threads return from the *onStart* handler and executes their transactions normally (line 5). If the thread is in *session* mode, then the thread's table entry

is updated to hold the current value of the *session* counter (line 4). Setting the *session* counter acts as a flag which other *session* mode threads can use to determine which transactions can be executed as part of their own *session*. Lines 6 to 11 perform the iterative execution of transactions. The thread first attempts to retrieve a new transaction to execute (line 7). If no more transactions are available, however, the thread calls an *onTimeout* handler (line 8). If the thread has not committed any transactions, it continues reading from the table. Otherwise, the thread breaks out of the loop and returns the *nocalltx* constant (line 12). If consensus has been reached (line 9) or the next transaction is successfully executed and no time remains (line 11), the thread breaks out of the loop and returns *nocalltx* (line 12).

OnPreCommit contains the code where *session* mode threads attempt to decide consensus. The *session*-mode thread invokes *compare-and-swap* (*CAS*) to set the status of the next entry in the UC (line 15). the thread updates its state, depending on the result of the *CAS* call (lines 16 and 18).

OnCommit is called after the *onPreCommit* handler has been invoked. If the calling thread is in *session* mode and it decided the consensus result, then it atomically increments the *session* counter (line 21) indicating to other threads that the *session* has terminated. In line 23, threads check the UC to determine whether their transaction was committed, and if so, the thread leaves *session* mode (line 24), otherwise the thread rolls-back execution and will attempt a new *session* (line 25).

OnAbort is invoked whenever any transaction aborts (see Algorithm 2, line 26). A flag is supplied to the abort handler to identify whether the abort was made implicitly (a concurrent conflict) or explicitly (a semantic conflict). In the case of explicit aborts, the aborting thread sets its *session* mode flag (effectively entering *session* mode).

OnTxSuccess is invoked when a transaction is successfully executed in *session* mode. The thread updates its bit mask (line 28), and decrements a private counter (line 29). If the counter has reached 0, the *onTimeout* handler is invoked (line 29). Threads invokes *onTimeout* (line 27) to determine whether they should continue executing transactions in the transaction table, or perform the *onPreCommit* handler (thus attempting consensus).

2.3 Session Locks

As with conventional *TinySTM*, locking is used to guarantee consistent reading and writing of shared data (*TinySTM* provides both read and write locks). To accommodate our CMP, however, we have added an extra type of lock, called a *session* lock, with the following properties:

- Once locked, a *session lock* grants access to a word of shared data for *any* thread operating in the same *session*, hence a *session lock* is locked only once per *session*;

– A *session lock* is never explicitly unlocked. A *session lock* has a viable lifetime for the duration of the *session* in which it was locked. Once the *session* has ended, the *session* lock is considered *stale* and may be removed at the discretion of any encountering thread.

Algorithm 2. *TinySTM* and Session Lock Handlers

```
   function onAbort(tx, explicit)
26   if explicit = true then tx.sessionMode ← true;
   function onTimeout(tx)
27   if commitCount(tx.txMask) > 0 then return 0;
     else return (tx.counter ← newLimit);
   function onTxSuccess(tx, txcall)
28   setBit(tx.txMask, txcall.id);
29   if decrement(tx.counter) = 0 then return onTimeout(tx);
30   return tx.counter;
   function onSharedAccess(tx, lock)
31   ctr ← sessionCounter;
32   if ¬tx.sessionMode then
33     if ¬sessionLocked(lock) then return proceed;
34     if ctr ≠ sessionNo(lock) then return stale;
35     tx.sessionMode ← true;
36     return killself;
     if consensusReached(sessionNo(tx.id)) then return killself;
     if ¬sessionLocked(lock) then return proceed;
37   if nextctr ≠ sessionNo(lock) then return stale;
38   return sessionLocked;
   function onLock(tx, lock, accessResult, accessType)
39   if ¬tx.sessionMode then
40     lockval ← createTinyStmLock(lock, accessType);
41     return (cas(lock.addr, lock.val, lockval) = success);
42   if accessResult = sessionLocked then return true;
43   nextctr ← sessionCounter;
44   sLockValue ← createSessionLock(nextctr);
45   return (cas(lock.addr, lock.val, sLockVal) = success);
```

In *TinySTM*, a lock is represented by a word-sized integer, with the value of the last two bits denoting the type of lock (binary 0 is unlocked, 1 denotes *write* locked and 2 denotes *read* locked). A *session* lock is represented by setting both bits. The remaining bits of the word value hold the *session* number in which the lock was set. Algorithm 2 (lines 31-45) shows two handlers which are invoked when dealing with *session* locks:

OnSharedAccess is called before a shared word is locked for reading or writing. Non *session* mode threads may attempt to lock shared data which is not *session* locked (line 33) or if the *session* lock is stale (line 34). Otherwise the thread enters *session* mode (line 35) and aborts (line 36). Threads in *session* mode, however, can attempt access of shared data as long as the *session* is still active.

OnLock is called whenever a thread attempts to lock shared data (lines 39-45). Non *session* mode threads create a normal *TinySTM* type lock and attempt

to lock the data (line 41) while *session* mode threads can immediately access *session* locked data (line 42). If the shared word is not *session* locked, then a *session* mode thread must attempt to lock the data (line 45).

3 Related Work

A range of CMPs currently exist but which can be categorised as either wait based and schedule-based. Wait-based CMPs [10,11] (e.g. *Greedy*, *Karma*, *Polka* etc), are typically trivial to implement, versatile and offer good performance. Heber et al, however, noted in [12] an inefficiency with wait-based approaches due to the difficulty in finding an adequate back-off period, given the dynamic nature of execution in STMs. Conversely, schedule-based CMPs typically reschedule or serialise aborted transactions. [13] exemplifies one such approach. Bai et al produced several 'transaction executor' models with the aim of equitably distributing transactions as 'jobs' among the threads of an application. 'Keys' are also used to predict the likelihood that conflicts will arise between executing transactions. Transactions which are likely to conflict are scheduled to be executed by the same 'worker' thread (thus enforcing serialisation).

CAR-STM [14] and Steal on Abort [15] are also schedule-based CMPs where transactional jobs are assigned to per-thread work queues. Both CAR-STM and Steal on Abort move aborted transactions to the work queues of conflicting transactions upon the occurrence of a conflict, to serialise the conflicting transaction's execution. Steal on Abort experiments with various techniques when rescheduling transactions among work queues. Additional work queues can also be created when the number of transactional jobs is high. *Hugh2* differs from the cited approaches of both wait-based and schedule-based CMPs, insofar as *Hugh2* is the only approach which focuses on the resolution of semantic conflicts. In addition, *Hugh2* requires a single transaction table to hold transactional jobs, but does not require the overhead of a thread pool to administer such jobs. *Hugh2* also explores multiple schedules in parallel during the process of contention management.

Similarly with *Hugh2*, several approaches to STM have been developed which rely on a Universal Construction (UC). Herlihy [16] introduced the UC concept to enable multiple threads to access shared data structures via a wait-free algorithm. Wamhoff [17] and Chuong [18] combined the UC technique with transactions to handle certain failure conditions. Crain et al have shown that it is possible to remove the abort semantics of STM using a UC [19]. While the cited approaches apply the UC technique for a STM system, *Hugh2* uses the UC for contention management.

Finally, TL-STM [20] is an adaptation of *SwissTm* which incorporates Thread-Level-Speculation (TLS) into memory transactions. TL-STM bears similarity to *Hugh2* insofar that platform parallelism is exploited to explore different permutations of transactional elements. More specifically, TL-STM seeks to enhance transactional throughput by reordering the internal execution elements of a transaction to better reflect concurrent schedules of execution. Conversely,

Hugh2 seeks to reorder whole transactions to accommodate semantic schedules of execution. Whereas TL-STM applies *internal reordering* based on the semantics of a transaction, *Hugh2* applies *external reordering* based on the semantics of an application.

4 Evaluation

In this section we present results from a series of benchmarks to demonstrate the performance of our system. The tests were carried out on a desktop PC featuring 2 x dual-core 3.07GHz Intel(R) processors with 4GB of RAM. The Operating System used was Ubuntu (Linux) version 13.04 and the Transactional Memory software was *TinySTM* version 1.04 with the Write-Back, Eager Transactional Locking scheme using visible reads. Experiments were carried out with increasing numbers of threads (from 2 to 16) with each run executed 5 times with the average results provided. Two existing CMPs were used as a measure of comparison with *Hugh2*, specifically *Karma* and *Polka* [11].

Two benchmarks were used to test the performance of *Hugh2*. The first scenario (*bank*) is provided in the *TinySTM* software and allows the execution of a number of transaction types on a set of simulated bank accounts. The 'bank' in this case is an array of account data structures. The second scenario (*vacation*) is part of the *STAMP* benchmark suite [7] and provides transactional accesses over several red-black trees to represent a holiday booking database system. Both scenarios provide update, read-all and write-all transaction types which can be generated at varying intensities. Transactions from the *vacation* scenario differ from the *bank* simulation insofar as they tend to execute more statements of greater complexity.

Semantic transactions were introduced into *bank* and *vacation*. In the *bank* scenario, two extra transactions (called *service charge* and *pay interest*) were created which explicitly call abort based on the balance of certain bank accounts. In the *vacation* scenario an additional red-black tree was created and two transaction types (called *create customer* and *remove customer*) which add and remove nodes while explicitly aborting if the contents of the tree is deemed incorrect. The semantic transactions introduce a consumer-producer relationship where a producer transaction should precede a consumer to grant mutual success. The semantic transactions interact with numerous other shared data elements, so it is expected that if semantic transactions must abort frequently, this activity will also increase the frequency of concurrent conflicts. Increasing the number of semantic transactions in a scenario means we can measure the impact of semantic conflicts on the application (for example, we might set up a scenario with 16 threads and specify that 8 of the threads execute semantic transactions to observe the effects of 50% semantic conflicts on the throughput of the application).

4.1 Transaction Throughput

Figure 2 provides graphs showing results for transaction throughput. Y-axes shows the number of transactions committed per second and X-axes show the

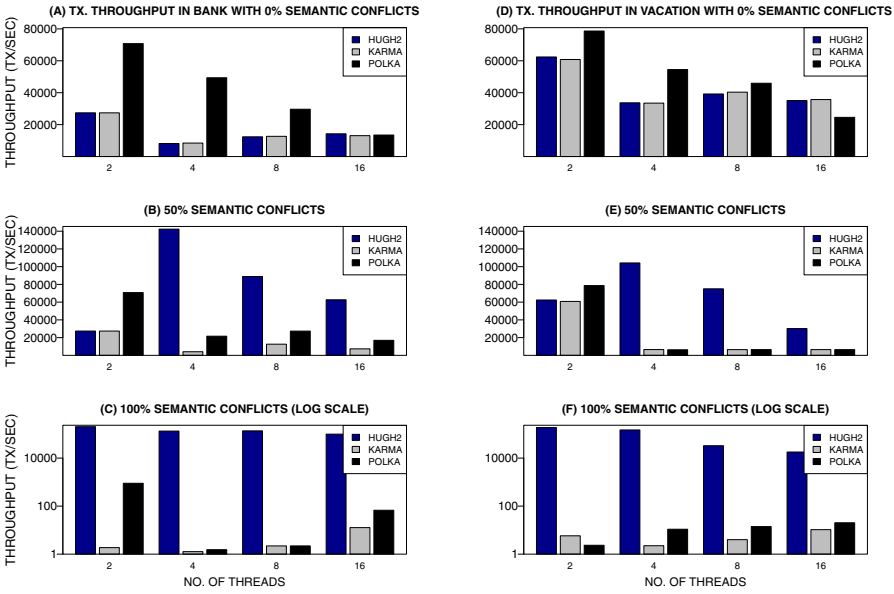

Fig. 2. Average transaction throughput for the Bank/Vacation scenario

number of threads used. Graphs 2(A) and 2(D) provide comparison between the *Karma*, *Polka* and *Hugh2* CMPs in the absence of semantic conflicts, for the *bank* and *vacation* scenarios respectively. The system which employs *Hugh2* for semantic conflicts, resorts to calling the *Karma* on occurrence of concurrent conflicts. As expected, with no semantic conflicts being generated in either graph, the performance of *Hugh2* and *Karma* are practically the same.

In Graphs 2(B) and 2(E) semantic conflicts have been introduced into both scenarios such that 50% of the threads generate semantic transactions in the case of thread numbers: 4, 8 and 16[1]. At this point the throughput for *Karma* and *Polka* have both fallen noticeably relative to the throughput for *Hugh2* which has increased substantially. In Graphs 2(C) and 2(F), semantic transactions are generated by 100% of the threads; once again the throughput for both *Karma* and *Polka* has reduced dramatically, whereas *Hugh2* outperforms both.

When comparing the *vacation* scenario to the *bank* scenario we can see that the *Polka* CMP mostly outperforms both the *Karma* and *Hugh2* CMPs when semantic conflicts are absent, and indeed, *Karma* CMP and has been cited as providing the best average performance of wait-based CMPs [15] (one notable exception, however, is in the *vacation* scenario when 16 threads are used). It is encouraging, however, to see that *Hugh2* can function in combination with an

[1] Two or more threads are required to resolve semantic conflicts (i.e. a producer and consumer). To show 50% semantic conflicts therefore requires at least four or more threads. The results for 2 threads show 0% semantic conflicts instead.

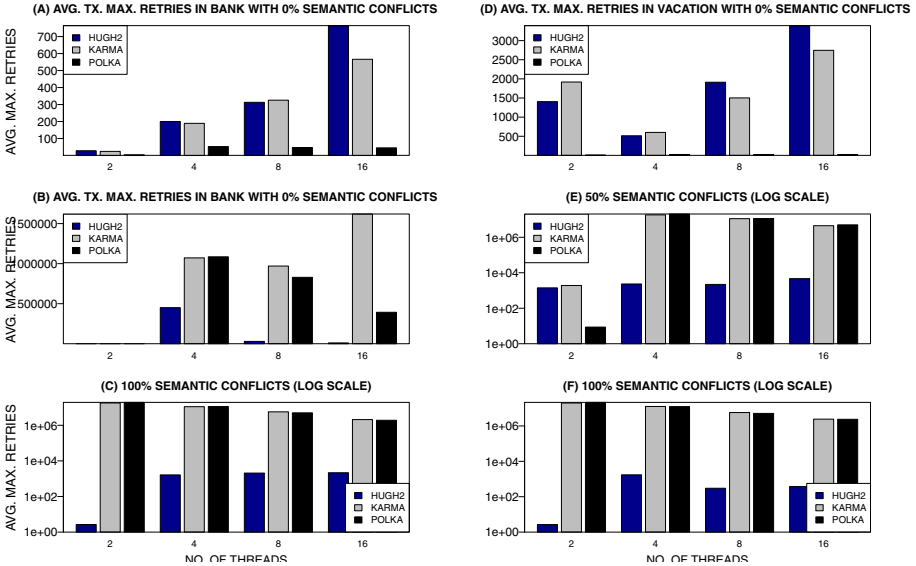

Fig. 3. Average maximum transaction retries for the Banking/Vacation scenario

existing CMP (in this case Karma), without degrading the performance with respect to resolving concurrent conflicts. Conversely, as semantic conflicts are introduced, neither *Karma* or *Polka* can approach the effectiveness of *Hugh2* in terms of transaction throughput. Although *Polka* almost always produces a higher throughput than *Karma*, neither approach maintains good performance when semantic conflicts are present (regardless of scenario). Throughput also diminishes for *Hugh2*, in the case of 50% semantic conflicts and to a lesser extent with 100% semantic conflicts. This suggests that the greater occurrence of threads producing concurrent conflicts has a negative impact on *Hugh2*.

4.2 Maximum Transaction Retries

Figure 3 presents results showing the average maximum retries for a transaction during the *bank* and *vacation* scenarios respectively. The format of the graphs in Figure 3 mirrors the previous results for transaction throughput with the exception that the Y-axis now shows average retries. A higher average number of retries is indicative of threads experiencing difficulty in resolving semantic conflicts. Hence, we expected that the average maximum retries should increase in tandem with an increase in semantic conflicts for the *Polka* and *Karma* managers, whereas this should not be the case for the *Hugh2* CMP.

Graphs 3(A) and 3(D) provide comparison between the *Karma*, *Polka* and *Hugh2* CMPs when no semantic conflicts are present. As expected, *Polka* CMP produces the smallest average maximum retries (graphs 2(A) and 2(D) have already shown that *Polka* produces the highest throughput in the absence of

semantic conflicts). In Graphs 3(B) and 3(E), however, semantic conflicts have been introduced at a rate of 50% (half the threads in the scenario generate semantic transactions in the case of thread numbers: 4, 8 and 16). A substantial increase in average maximum transaction retries is now observable in all CMPs, although *Hugh2* produces the best performance.

In Graphs 3(C) and 3(F), semantic transactions are being created by 100% of threads. Once again the average maximum number of retries has increased for both *Karma* and *Polka* CMPs. In the case of *Hugh2*, the average maximum has fallen, with neither *Karma* or *Polka* tackling semantic conflicts more effectively than *Hugh2*. In addition, there is only a negligible difference in performance between *Polka* and *Karma* (suggesting that neither policy is more effective at resolving semantic conflicts).

5 Conclusion

This paper presents *Hugh2*, a CMP which deals with semantic conflicts via the speculative execution of aborted transactions. We have described how *Hugh2* can be integrated with a word based STM using a new *session* locking mechanism. Two substantial benchmarks demonstrated performance improvements once semantic conflicts are introduced. Given that *Hugh2* can be incorporated with any existing CMP, it would be interesting to test the performance of *Hugh2* against a wider range of CMPs. In addition, incorporating semantic conflicts into the remaining STAMP benchmarks may be useful in order to observe how semantic conflicts affect a diverse range of scenarios.

Going forward, we believe the *session* lock mechanism raises some exciting possibilities for exploring our work within a distributed STM application. In particular, *session* locks may provide a greater scalability in the context of DSTM, given *session* locks may be shared across threads and need not be explicitly unlocked.

References

1. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 48–60. ACM (2005)
2. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 284–298. Springer, Heidelberg (2006)
3. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. In: ACM SIGPLAN Notices, vol. 41, pp. 253–262. ACM (2006)
4. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 237–246. ACM (2008)

5. Dragojević, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. In: ACM Sigplan Notices, vol. 44, pp. 155–165. ACM (2009)
6. Felber, P., Fetzer, C., Marlier, P., Riegel, T.: Time-based software transactional memory. IEEE Transactions on Parallel and Distributed Systems 21(12), 1793–1807 (2010)
7. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: Stamp: Stanford transactional applications for multi-processing. In: IEEE International Symposium on Workload Characterization, IISWC 2008, pp. 35–46. IEEE (2008)
8. Sharp, C., Morgan, G.: Hugh: A semantically aware universal construction for transactional memory systems. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 470–481. Springer, Heidelberg (2013)
9. Sharp, C., Morgan, G.: Introducing semantic conflict resolution to word based software transactional memory. Technical report, 10 p. Newcastle University, UK (2014)
10. Guerraoui, R., Herlihy, M., Pochon, B.: Towards a theory of transactional contention managers. In: Annual ACM Symposium on Principles of Distributed Computing: Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing, vol. 23, pp. 316–317 (2006)
11. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 240–248. ACM (2005)
12. Heber, T., Hendler, D., Suissa, A.: On the impact of serializing contention management on stm performance. Journal of Parallel and Distributed Computing (2012)
13. Bai, T., Shen, X., Zhang, C., Scherer, W., Ding, C., Scott, M.: A key-based adaptive transactional memory executor. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1–8. IEEE (2007)
14. Dolev, S., Hendler, D., Suissa, A.: Car-stm: Scheduling-based collision avoidance and resolution for software transactional memory. In: Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing, pp. 125–134. ACM (2008)
15. Ansari, M., Luján, M., Kotselidis, C., Jarvis, K., Kirkham, C., Watson, I.: Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In: Seznec, A., Emer, J., O'Boyle, M., Martonosi, M., Ungerer, T. (eds.) HiPEAC 2009. LNCS, vol. 5409, pp. 4–18. Springer, Heidelberg (2009)
16. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(1), 124–149 (1991)
17. Wamhoff, J., Fetzer, C.: The universal transactional memory construction. Technical report, 12 p. University of Dresden, Germany (2010)
18. Chuong, P., Ellen, F., Ramachandran, V.: A universal construction for wait-free transaction friendly data structures. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, pp. 335–344. ACM (2010)
19. Crain, T., Imbs, D., Raynal, M.: Towards a universal construction for transaction-based multiprocess programs. In: Bononi, L., Datta, A.K., Devismes, S., Misra, A. (eds.) ICDCN 2012. LNCS, vol. 7129, pp. 61–75. Springer, Heidelberg (2012)
20. Barreto, J., Dragojevic, A., Ferreira, P., Filipe, R., Guerraoui, R.: Unifying thread-level speculation and transactional memory. In: Narasimhan, P., Triantafillou, P. (eds.) Middleware 2012. LNCS, vol. 7662, pp. 187–207. Springer, Heidelberg (2012)